CS3001, Algorithm Design and Analysis
Example Exam Questions

1) a) The design paradigm "Dynamical Programming" is used when an algorithm recurses to several subproblems, which in turn again recurse to several subproblems such that many of these subproblems of the second recursion level are the same. Instead of solving these problems again and again, solutions are stored once they are computed. If the same subproblem arises again, the stored solution can be fetched immediately. This reduces the (often exponential) runtime of the recursive algorithm to a runtime proportional to the number of *different* subproblems which arise. (As the purely recursive approach will encounter each of these subproblems at least once, this cannot be worse.)
The gain in runtime however is paid for by the increased storage required for the solutions of the subproblems. It can therefoe only be used if the total number of subproblems is not too big.
b) Suppose there is a solution. If this solution does not use the n-th item it must be a solution for the knapsack problem with $n-1$ items. If it uses the n-th item, we can consider the remaining knapsack size $K - s_n$ which must be filled with $n-1$ items. So there must be a solution for a smaller Knapsack problem. Denote by $A(n, K)$ the knapsack problem with the first n items and a knapsack of size K. To find a solution for $A(n, K)$ we therefore either need a solution to $A(n-1, K)$ or a solution to $A(n-1, K-s_n)$. This yields a recursive algorithm. We use dynamical programming to avoid solving the same subproblems again.
Now give the knapsack algorithm as in the lecture. Explain what the variables stand for and how to get the solution from the data structures built by the algorithm.

2) a) We use a heap in which we store the entries of the sequence together with their count. The algorithm first stores all integers in the heap and then fetches the entries from the heap in sorted order:

```
heap := [];
for i in input do
  if i ∈ heap then
                {cost: O(depth of tree)}
    increment count;
  else
    insert i with count 1;
  fi;
od;
while heap not empty do
  remove top element i from heap;
  output r;
od;
```

If we use a binary tree to implement the heap, the tree has depth $\log \log n$ (because there are $\log n$ different numbers). Therefore one insert or search operation takes $O(\log \log n)$ element comparisons and building the heap thus take $O(n \log \log n)$ comparisons. Taking the top element from the heap (preserving the heap structure) also takes $O(\log \log n)$ steps, so the output stage has runtime $O(n \log \log n)$ as well.
b) This bound holds for arbitrary sets of objects. This problem has only $\log n$ different numbers and is therefore a special case. (An extreme would be if all elements are known to be the same, then sorting becomes trivial.)

3) a) Denote the time taken for a $n \times n$ matrix multiplication by $T(n)$. Then 7 products of $\frac{n}{2} \times \frac{n}{2}$ matrices take time $7T(\frac{n}{2})$. A matrix addition can be implemented by a double **for**-loop like this:

for $i \in [1..n]$ **do**
  for $j \in [1..n]$ **do**
    $a_{i,j} := b_{i,j} + c_{i,j}$;
  od;
od;

Therefore $T(n)$ fulfills the recurrence relation $T(n) = 7T(\frac{n}{2}) + 18An^2 + B$, where A and B are constants which represent the time taken by function calls, setup of for loops and similar initializations. As shown in the lecture (Master Recurrence relation) such a recurrence relation fulfills:

$$T(n) = O\left(n^{\log_2(7)}\right) \cong O\left(n^{2.81}\right)$$

(because $7 > 2$) *(You are not required to evaluate* $\log_2(7)$*.)*
b) We subtract $T(n)$ from $T(n+1)$:

$$T(n+1) - T(n) = n + 1 + \sum_{i=1}^{n} T(i) - n - \sum_{i=1}^{n-1} T(i) = T(n) + 1$$

This yields the ordinary recurrence: $T(n+1) = 2T(n) + 1$. This yields (give a few expansion steps to show the pattern):

$$\begin{aligned} T(n) &= 2T(n-1) + 1 = 4T(n-2) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 \\ &= \cdots = 2^{n-1}T(1) + 2^{n-2} + \cdots + 2 + 1 = \sum_{i=0}^{n-1} 2^i = 2^n - 1, \end{aligned}$$

using the summation formula for the geometric series.

4) a) *Give the definitions/statements from the lecture.*
b) Suppose l is a leaf of a tree which is contained in a vertex cover U. There is only one edge adjacent to l, call it $e = \{l, m\}$. If we remove l from U and add m to it, we get a set W of vertices which still covers e. It also covers all other edges. Therefore W is a vertex cover with $|W| = |U|$. We can iterate this process and obtain a vertex cover of the same size (thus still minimal) which contains no leaves.
c) Take a tree T. If all vertices are leaves, T contains only two vertices and one vertex is a cover. Otherwise take a vertex v of T which is not a leaf but adjacent to a leaf l. Because of b), there is a minimal vertex cover U of T which does not contain l. To cover the edge $\{v, l\}$ it must contain v. Removing v and all adjacent leaves yields a smaller tree T'. The set $U \setminus \{v\}$ covers the edges not adjecent to v and thus is a vertex cover for T'. Viece versa any such vertex cover for t' together with v is a cover for T.
This yields the following algorithm:

function cover(T)
  if $|T| = 2$ **then** return one vertex set **fi**;
  find a vertex v adjacent to a leaf.
  let T' the reduced tree.
  $X =$ cover(T').
  return $X \cup \{v\}$.

The algorithm can recurse at most $|V|$ times, the remaining time spent in one call of "cover" is constant. Therefore the runtime is bounded by $O(|V|)$.